UML2Alloy Reference Manual

Version: 0.26

Date: Monday 10th September, 2007

Kyriakos Anastasakis

The University of Birmingham
School of Computer Science

Contents

List Of Figures iii

1 Introduction **1**

1.1 Minimum Requirements 1

2 Translating UML **2**

2.1 Classes, Subclasses and Types 2

2.2 Enumerations 2

2.3 Associations 3

2.4 Class Attributes 3

2.5 Class Methods 4

3 Translating OCL **5**

3.1 Invariants 5

3.2 Operations 6

3.3 OperationCallExps 6

3.3.1 Implies 6

3.3.2 And 6

3.3.3 Dot 6

3.4 If 6

3.5 Iterator Expressions 7

3.6 notEmpty 7

3.7 Not 7

3.8 Variable Expressions 7

3.9 Integer Expressions 7

4 UML2Alloy: Usage Manual **8**

4.1 Installation 8

4.2 Using UML2Alloy 8

4.2.1 Step 1 8

4.2.2 Step 2 8

4.2.3 Step 3 9

5 Examples **10**

5.1 Two Unidirectional vs One Bidirectional Associations . . 10

5.2 Using UML2Alloy in a model with integer expressions . . 12

5.2.1 The UML model 12

5.2.2 Transformation 14

5.2.3 Results 14

A XMI Compatibility **19**

B UML2Alloy Manual Document Changelog **20**

C UML2Alloy Application Changelog **21**

References 22

List of Figures

4.1 First Screen of UML2Alloy 8
4.2 Second Screen of UML2Alloy 9
4.3 Final Screen of UML2Alloy 9

5.1 Example UML model of two classes connected with two unidirectional and one bidirectional association 10
5.2 Exporting model to XMI using ArgoUML 11
5.3 Selecting the mapping 11
5.4 Selecting the statement we want to check 12
5.5 Counterexample 13
5.6 UML model of example 13
5.7 Screenshot of the second step of the transformation process 14
5.8 Providing the scope 15
5.9 Alloy model of example with integer expressions 16
5.10 A counterexample to the assertion 17

1 Introduction

This document is the user manual of UML2Alloy version 0.26 alpha [2]. The transformation rules are based on Alloy version 4 and UML version 2.0.

UML2Alloy is the outcome of research which attempts to formalise UML using Alloy [4]. UML2Alloy translates UML Class Diagrams enriched with OCL constraints to Alloy. Alloy models can be automatically analysed using the Alloy Analyzer [1]. The current version of UML2Alloy performs the translation to the Alloy language and creates a text file with the Alloy model. The user needs to use the Alloy Analyzer to open the text file and perform the analysis using Alloy Analyzer. In the future we endeavour to use Alloy Analyzer as a library to incorporate its functionality in UML2Alloy.

The Alloy Analyzer works by translating the model into a boolean expression, which is analysed by SAT solvers embedded within the Alloy Analyzer. A user-specified scope on the model elements bounds the domain, making it possible to create finite boolean formulas for the evaluation by SAT solvers. The Alloy Analyzer offers two analysis methods. One is *simulation* and the other is *assertion checking*.

Simulation produces a random instance of the model that conforms to the constraints. The absence of an instance implies that most probably the model is inconsistent. It is possible from within the Analyser, to narrow down the statements, which produce the inconsistent model. This functionality is called UnSat Core.

Assertion checking, checks whether an assertion, which is a statement which should follow from the specification, is satisfied by the model. If it is not, a counterexample is presented to the user. A counterexample is an instance of the model, which violates the assertion.

This manual contains some information for both Alloy and the Alloy Analyzer for people familiar with UML but unfamiliar with Alloy. For more information on the Alloy Analyzer and the Alloy language, the reader is refereed to [4] and [1].

If the reader is interested in the transformation rules and the restrictions on the translation, the reader is referenced to Chapters 2 and 3. If the reader wants to learn details on how to use UML2Alloy, (s)he is referenced to Chapter 4. Finally Chapter 5 presents a number of simple examples to demonstrate how UML2Alloy can be used.

1.1 Minimum Requirements

UML2Alloy requires Sun Java 1.5

2 Translating UML

This section provides a brief informal overview of the transformation rules from UML to Alloy. The rules are implemented in UML2Alloy. This section also mentions any restrictions that might exist.

2.1 Classes, Subclasses and Types

- ◯ Translates Classes to Signatures. If the class *isAbstract* it will be translated to an abstract Signature.
- ◯ Translates subclasses to subsignatures. The default subclassing semantics is {incomplete, disjoint} [7, p. 72]
- ◯ Enumerators are translated to abstract Signatures and Enumerator literals are translated to subsignatures with one element (Singletons). For more information see Section 2.2.
- ◯ Custom data types. We translate all Classes with the "DataType" stereotype to Signatures. In principle Custom data types are not treated any different than any other Class.

Restrictions

- X No primitive data types (as defined in the UML standard) are currently supported by the implementation. (Since UML2Alloy 0.3, rudimentary support for integer expressions has been introduced).
- X Multiple packages are not supported. Translation of the classes of one package at a time takes place.
- X Multiple inheritance is not allowed (the transformation rules have been defined, but not implemented in the tool).
- X Only subclasses with {incomplete, disjoint} semantics are allowed. Again the rules for the rest of the cases have been defined, but we haven't implemented them.
- X Subclasses can not redefine attributes, operations. Thus attribute and operation names can not have the same name as the attribute and operation names of the superclasses.
- X The NameSpace to which Classes can belong to is not restricted by the UML specification. Thus a Class can belong to the NameSpace of another Class (*inner Classes*). Our rules only support Classes which belong to the NameSpace of a Package.

2.2 Enumerations

- ◯ Enumerations are translated to abstract Signatures.
- ◯ Enumeration literals are translated to Subsignatures, which extend the abstract Signature of the Enumeration. Since each enumeration literal represents only one value, it is declared as a Signature represented by one atom in Alloy.

Restrictions

X Enumerations have to be defined in the UML class diagram as Classes with an "Enumeration" stereotype. Enumeration literals have to be defined as attributes of the Class. Attributes have to be of *void* type.

2.3 Associations

Please note that in this section the term *Association End* is used, to reference the ends of an association. Even though this term is used in the UML specification (for example see [7, p. 14] or [7, p. 40]), the UML metamodel reffers to Association Ends as *Properties* [7, p. 29].

◯ All association ends *have* to have names. Role names have to be unique. (Since UML2Alloy 0.3 this is not a requirements). If an Association End does not have a name, it will be assigned the name of the

◯ Unidirectional Associations: The navigable Association End appears as a field in the Signature to which the original Class the association belonged to, was translated.

◯ Bidirectional Associations: Association Ends appear as fields in both the Signatures that participate in the relation. An additional fact is added to the model, to denote that the relations are symmetric.

◯ Multiplicities: Depending on the multiplicity of the Association Ends, the necessary multiplicity facts will be added to the model.

Restrictions

X Association End names have to be *unique* throughout the model (i.e. all Association Ends have to have unique names, in contrast with the UML specification, which allows two separate Association Ends to have the same name, as long as they belong to different Classes).

X Association End annotations are not supported (e.g. *subsets*, *ordered* etc.).

X Only binary associations are supported in the current version of the implementation.

X Custom multiplicities are not currently supported by the implementation. Only the predefined (1, 0..1, 1..* and 0..*) multiplicities are supported.

X The UML specification allows for unspecified navigability [7, p. 41]. Such cases are not supported.

X The UML specification allows for binary associations with both ends non navigable. Again such cases are not supported.

X This version of UML2Alloy has not implemented our rules for aggregation and composition. Associations with composition or aggregation have to be refactored using the method defined by Gogolla and Richters [3].

X Association Classes are not currently supported.

2.4 Class Attributes

◯ Class attributes are translated to Signature fields.

Restrictions

X Attributes have to be of a specific type (i.e. not *void*). This restriction does *not* apply for enumeration literals, see Section 2.2.

X Only attributes with multiplicity of 1 are supported.

2.5 Class Methods

◯ Class methods are translated to Alloy predicates. For more information see Section 3.2.

Restrictions

X Class method can not return any type, other than Boolean or Void.

X Operations in UML may raise exceptions. UML2Alloy does not deal with such cases.

3 Translating OCL

This section presents the details on the translation of OCL statements (invariants and pre and postconditions) to Alloy.

IMPORTANT NOTE: NO shorthand notation is supported for OCL expressions. For example:
Use: collection → forAll(v:Type \| boolean-expression-with-v)
Do NOT use: collection → forAll(v \| boolean-expression-with-v)
Do NOT use: collection → forAll(boolean-expression)

Use: self.attribute
Do NOT use: attribute

OCL expressions which reference the Boolean values *true* or *false* directly are not supported. For example an expression like:

```
if condition then
    expression
else
    false
endif
```

is not currently supported. Instead an expression which always evaluates to false could be used, in the '*else*' branch of the above statement.

3.1 Invariants

Please NOTE: The UML standard specifies that invariants are implicitly qualified over all of the instances of the context [5, A.3.1.5]. So for example an invariant:
context C1
inv test: self.attribute = value

is implicitly transformed to:

context C1
C1.allInstances → forAll(v1:C1 \| v1.attribute = value)

UML2Alloy currently supports only the second expanded form (i.e. with the explicit quantification, using allInstances and forAll), NOT the first form. Support for the first form will be provided in the future.

- ◯ Class invariants are generally translated to Alloy facts. They can also be translated to Assertions. This is defined during the transformation process. For more information please see the examples in Section 4.2.2.
- ◯ Each invariant is translated to a predicate and then a fact statement, which references the predicate is added.

Restrictions

X Only Class invariants defined in the namespace of a Class are supported (i.e. Class invariants).

X Invariants have to have a name.

3.2 Operations

The effects of Operations is defined using OCL *pre* and *post* conditions.

NOTE: The current version of the implementation only supports postconditions.

◯ Operations are transformed to Alloy Predicates. Operations are defined in the context of a Class. Because in Alloy Predicates belong to the NameSpace of the model, an instance of the Signature on which the specification of the Predicate is applied, is passed as a parameter to the Class. Therefore all references to the *self* keyword used in OCL are transformed to the parameter passed in the Predicate. For a detailed example please see Section[].

Restrictions

X The current implementation supports one OCL expression as specification to the Operation.

X Pre and Post conditions have to have a name.

X Operations can be of either Boolean or Void type.

X Currently Operations can not have any parameters.

3.3 OperationCallExps

In OCL a large number of expressions are OperationCall expressions. For example *equality*, *implies*, *and*, *or*, etc. expressions, are considered to be OperationCall expressions. The following shows how those OperationCall expressions are translated to Alloy.

3.3.1 Implies

◯ Implies OperationCalls are directly translated to an Alloy implies formula (*ImplicationFormula*).

3.3.2 And

◯ An OCL *and* OperationCall is translated to an Alloy binary formula (*BinaryFormula*) with a AND Logical operator.

3.3.3 Dot

Dot OperationCall expressions in OCL, are navigation expressions, which use the navigation dot (.).

◯ An OCL *Dot* OperationCall is translated to an Alloy binary expression *BinaryExpr* with a JOIN_DOT binary expression Operation.

3.4 If

◯ An OCL *If Expression* is transformed to an Alloy implies formula (*ImplicationFormula*). In Alloy an ImplicationFormula can have an *else* clause, which is what the OCL *else* clause is translated to.

3.5 Iterator Expressions

◯ An OCL Iterator Expression is transformed to an Alloy *Quantified Formula*.

Restrictions

X Currently only the *forAll* and *exists* Iterator Expressions are supported.

X Can only translate Iterator Expressions of the form: *Class.allInstances()* → *forAll* or *Class.allInstances()* → *exists*.

X Do not currently support nested Iterator Expressions.

3.6 notEmpty

◯ An OCL notEmpty expression is transformed to an Alloy *ElemIntFormula*. In particular it is translated to an Alloy Formula which enforces the size of the Expression to be greater than zero. For an example please refer to Chapter 5.

Restrictions

X The *notEmpty* operator has to be applied to a simple navigation expressions such as: *Person.allInstances rightarrow exists(p:Person | p.wife → notEmpty)*. Currently UML2Alloy does *not* support the application of *notEmpty* to more complicated expressions, such as Iterator Expressions.

3.7 Not

◯ OCL *not* expressions (expressions with a negation) are directly transformed to Alloy Negation Formulas *NegFormula*.

3.8 Variable Expressions

◯ OCL *Variable Expressions* (for example in the expression '*self.name*', *self* is a Variable Expression), are transformed to Alloy Variable Expressions (*VariableExpr*).

3.9 Integer Expressions

◯ UML Class attributes of type *integer* are transformed to fields of type *Int*.

◯ OCL integer OperationCall expressions that match the pattern *expression IntCompOp LiteralInteger*, where *expression* is an expression, *IntCompOp* is an integer comparison operation (i.e. =, <, >, >=, <=) and a *LiteralInteger* is an expression that denotes to an integer value (i.e. -3,0,1, etc.), are transformed to Alloy *ElemIntFormula*s.

◯ OCL *LiteralIntegers* [6, p.60] are transformed to Alloy *LiteralIntExpr*s.

Restrictions

X Currently (version 0.3) do not support operations (i.e. addition, subtraction, multiplication, division) over integer values.

X Currently (version 0.3) only support OCL expressions that follow the pattern: *expression IntCompOp LiteralInteger*, where *expression* is an expression, *IntCompOp* is an integer comparison operation (i.e. =, <, >, >=, <=) and a *LiteralInteger* is an expression that denotes to an integer value (i.e. -3,0,1, etc.).

4 UML2Alloy: Usage Manual

This chapter is a brief guide on how to use UML2Alloy. It is suggested that ArgoUML version 0.19.5 is used to develop any models. The reason for this is that ArgoUML provides syntax checking for OCL statements. Please note that apparently some versions (greater than 0.19.5) have a bug in the OCL syntax checker.

4.1 Installation

Unpack the compressed file of the distribution to a directory. Depending on the platform edit the *run_windows.bat* or *run_linux.sh* files and make the necessary changes to the path variables.

The program can be started by executing the *run_windows* or *run_linux* files respectively.

4.2 Using UML2Alloy

4.2.1 Step 1

Figure 4.1 depicts the first screen that prompts the user to select the path to the XMI file of the UML model. For a table of the XMI files currently supported by UML2Alloy, please refer to Appendix A. The user then needs to select the path to the Alloy file (text file with an *als* extension) that will be generated if the transformation is successful.

The user then needs to click on *Parse*. This starts the parsing method. Messages on the progress appear on the console. The next screen that will be presented to the user is depicted in Figure 4.2.

4.2.2 Step 2

In this screen the user is presented with all the Invariants and Operations of the UML models. The first column depicts the Operation or Invariant name. The second column shows if the statement is an Invariant or Operation in the original UML model. Finally the third column has a drop down menu, which allows the user to select to what kind of Alloy statement the original UML statement will be transformed.

The drop down menu with the selections is populated with the following rules. An OCL Invariant can be translated to an Invariant (*Fact* in Alloy terms), *Assertion* or *Simulation* statement. An Operation

Figure 4.1: First Screen of UML2Alloy

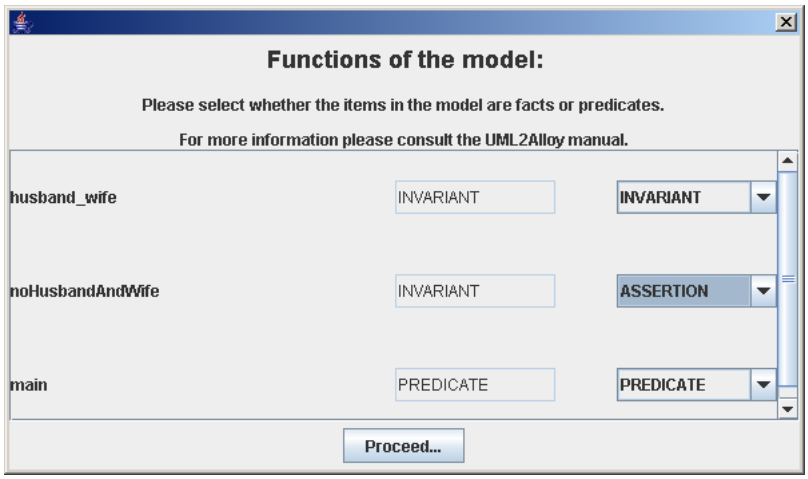

Figure 4.2: Second Screen of UML2Alloy

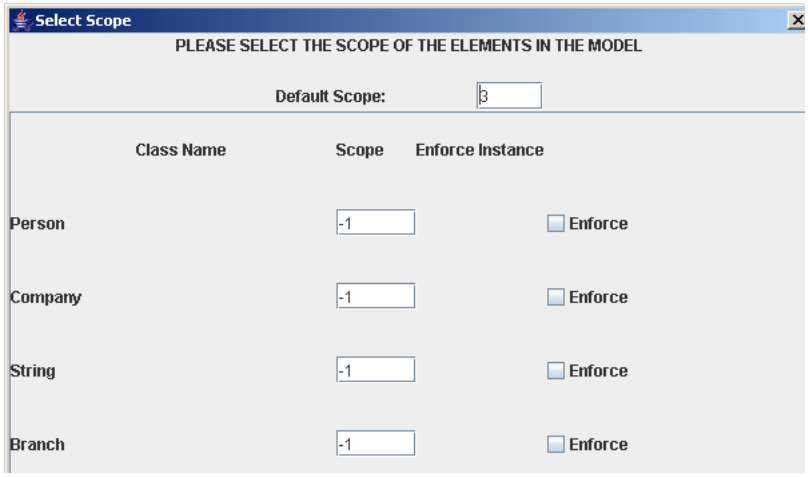

Figure 4.3: Final Screen of UML2Alloy

can be translated to a Predicate, Simulation or Invariant. An Operation with an OCL statement, which uses the '*self*' keyword, can only be transformed to a Predicate.

4.2.3 Step 3

The last step towards the translation of UML to Alloy, requires the user to specify the Scope. Figure 4.3 depicts a screenshot of this last step. The user can specify the default scope, which is the scope that will be applied to all model elements. This can be overridden for specific model elements if a value other than -1 exists in the *scope* box of a specific model element. By checking the checkbox of the last column the user can *enforce* that a model element will have instances in the model. For a concrete example on the usage of this function, please refer to the examples in Chapter 5.

5 Examples

This chapter presents a number of simple examples to demonstrate the usage of UML2Alloy.

5.1 Two Unidirectional vs One Bidirectional Associations

In this example we want to assert if two unidirectional associations between two classes have the same meaning as one bidirectional association. To do so, we generate the model depicted in Figure 5.1. The model was developed using ArgoUML version 0.19.5.

The model does not have any constraints. We want to assert that the two unidirectional associations between Class1 and Class2 have the same effect on the model as the bidirectional association between Class3 and Class4. The assertion we use is stated in OCL:

```
-- We call this statement 'areSame'
context Class1
inv areSame : Class1 . allInstances() -> forAll (
cl1: Class1 | cl1 . c2 . c1 = cl1 )
```

This assertion states that for each Class1, if we navigate to Class2 and come back we will end up in the same object as where we started.

In ArgoUML we define this assertion as an *Invariant* and when we use UML2Alloy in Step 2, we define that we want to translate it as an assertion, i.e. a statement to check.

Similarly we formulate the following OCL statement to reason about the association between Class3 and Class4.

```
-- We call this statement 'bidirectional'
    context Class3
    inv bidirectiional : Class3 . allInstances -> forAll (
    cl3 : Class3 | cl3 . c4 . c3 = cl3 )
```

We now export the model to XMI format from within ArgoUML. Figure 5.2 depicts a screenshot that shows how to achieve that in ArgoUML.

Next we invoke UML2Alloy. In the first step of the transformation process we select the path to the XMI file and the Alloy file we want to create. In the next step we select what each OCL statement maps

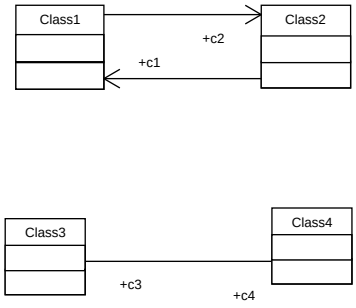

Figure 5.1: Example UML model of two classes connected with two unidirectional and one bidirectional association

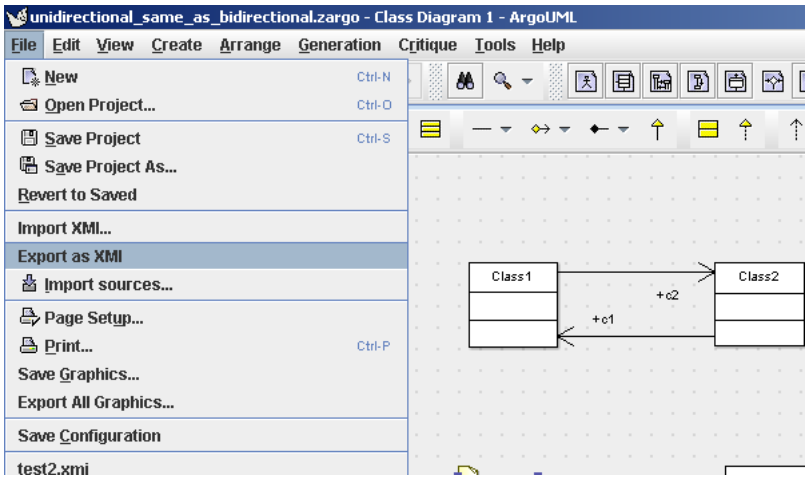

Figure 5.2: Exporting model to XMI using ArgoUML

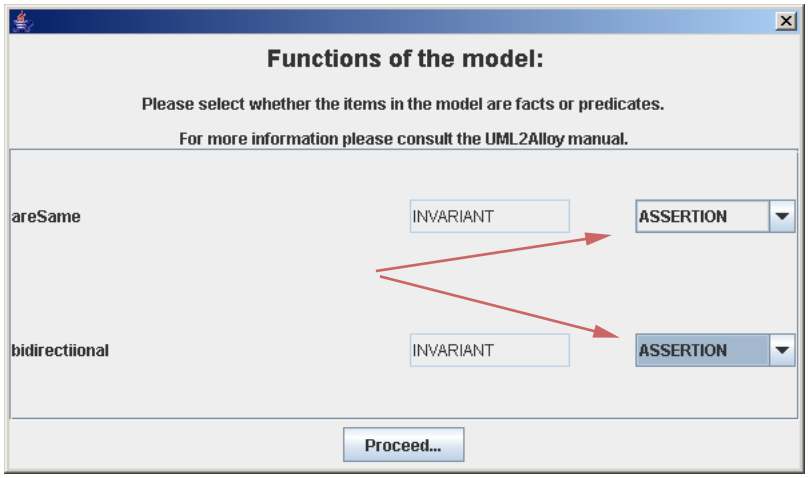

Figure 5.3: Selecting the mapping

to in Alloy. This step is depicted in Figure 5.3. The *areSame* and *bidirectional* are the two invariants we specified in ArgoUML. By default the statements specified as invariants in the original UML model are translated to invariants in Alloy (*facts*, according to the Alloy terminology). We want to check if these statements are satisfied. They therefore need to be transformed to assertions, not invariants. We specify that by selecting in the drop down menus that the original OCL invariants are transformed to assertions (indicated by the red arrows in Figure 5.3). We then click on the *proceed* button.

Finally in the last step, we specify a default scope of 3. We do not need to change any other parameters in the last screen. The transformation process has finished without an error and the Alloy file has been generated in the path we specified in the first step.

We now need to invoke the Alloy Analyzer to conduct the analysis. On the main Alloy Analyzer window, we *Open* the Alloy file generated by UML2Alloy. We first have to *Build* the file. Then we need to select from the drop down menu (indicated by the red arrow in Figure 5.4) the statement we want to simulate or check. In this case our model has only two assertions, *areSame* and *bidirectional*.

We first select from the drop down menu the '*areSame*' assertion and press on *Execute*. The Alloy Analyzer produces a *counterexample*. This is an instance of the model that violates the assertion. This instance is depicted in Figure 5.5.

From a closer examination of the instance we can see that there is a class, *Class1_0* related to *Class2_0*,

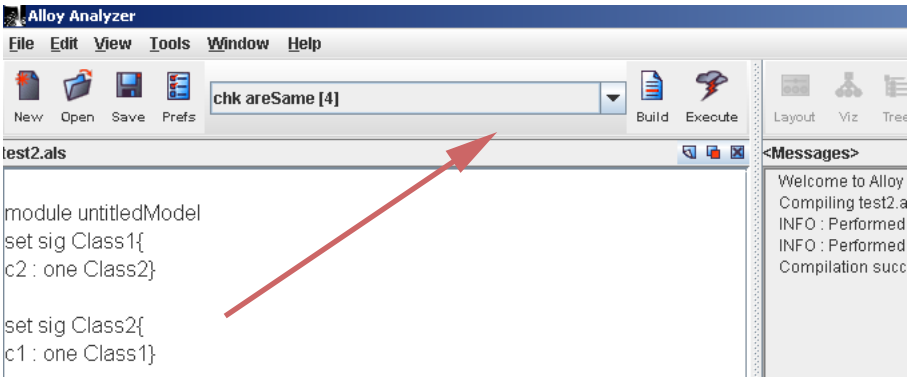

Figure 5.4: Selecting the statement we want to check

which in turn is related to *Class1_1*. The assertion therefore fails, because navigating from a class to another class and back, we end up with different class.

Currently the results of the analysis are represented using the Alloy Analyzer representation form (for example as a Tree, as shown in Figure 5.5). In the future we plan to represent the results of the analysis in terms of UML (i.e. object diagrams).

Checking the *bidirectional* assertion does not provide any counterexamples.

For more information on the Alloy Analyzer, please consult the Alloy website [1].

5.2 Using UML2Alloy in a model with integer expressions

This section presents a simple example, where a UML model with integer expressions is transformed to Alloy.

5.2.1 The UML model

Figure 5.6 depicts a simple UML model. There two classes. A *Person* class and an *Adult* class, which extends the Person class. The *Person* class has an attribute *age* of type *Integer*. An *Adult* can *marry* another Adult.

There are some OCL constraints (related to integer values) attached to the model. The constraints of the *Person* class are:

```
context Person
inv ageInv :
Person . allInstances() -> forAll (
p : Person | p . age > 0 )
```

The constraints of the *Adult* class are:

```
context Adult

inv adultInv :
Adult.allInstances ->forAll(
ad:Adult | ad . age >= 18 )
```

We need to check that in such a model a Person over 18 can only marry only another Person whose age is over 18. This *assertion* can be formulated in OCL:

```
context Adult
inv assertion :
Adult . allInstances -> forAll (
```

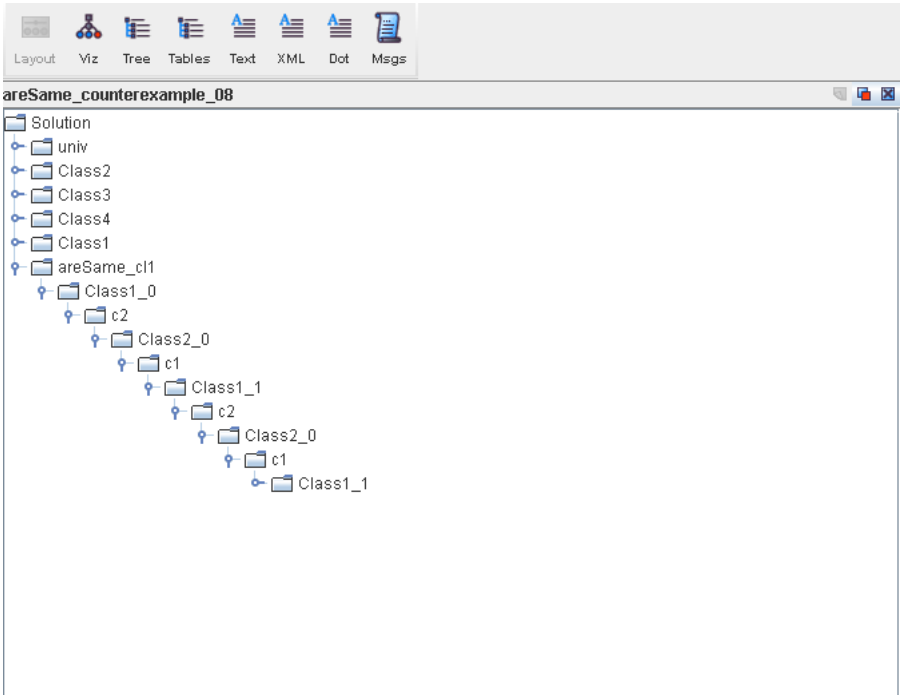

Figure 5.5: Counterexample

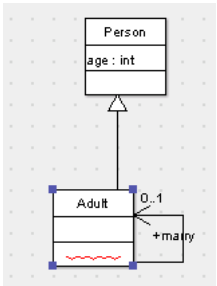

Figure 5.6: UML model of example

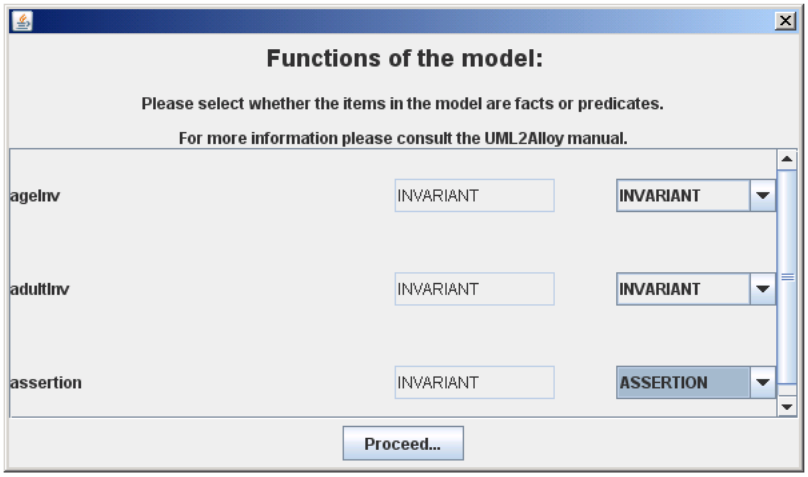

Figure 5.7: Screenshot of the second step of the transformation process

```
ad : Adult | ad . marry . age >18 )
```

We now export this model from our UML modelling environment to XMI format and use UML2Alloy to transform it to Alloy.

5.2.2 Transformation

In the first step of the transformation process we provide the path to the XMI file that contains the UML model we presented in the previous section. We also specify the path of the Alloy file that will be created.

In the second step of the transformation process we specify how invariants and operations of the UML model map in Alloy. Figure 5.7 presents a screenshot. This shows that the *ageInv* and *adultInv* will be transformed into Invariants (i.e. *Fact*s in Alloy), while the *assertion* statement, will be transformed to an *assertion* in Alloy. The next step in the transformation process is to specify the scopes.

As depicted in Figure 5.8, when defining the scope we now have another option, the '*Integer Values Range*', which is a drop down menu. Like all elements in Alloy, we need to specify a scope for integer expressions. The user can select from the drop down menu the range of integer expressions for which the model will be probed for a solution or a counterexample. In this case we have selected a range of -32,31. So the largest negative integer in the model is -32, while the largest positive integer is 31.

The scope of integer values can be chosen according to the biggest positive/negative value we expect integers expressions to have to in the model. For example in our example we care about integer expressions of scope 18 and over. This is why we chose a scope of -32,31 for this example.

Figure 5.8 also shows that we have enforced that there will be instances of Persons and Adults in the model. We have also specified a default scope of 4, but defined a specific scope of 8 Persons.

5.2.3 Results

The Alloy model that was produced by UML2Alloy is depicted in Figure 5.9. Executing this Alloy model provides a counterexample. The counterexample is depicted in Figure 5.10.

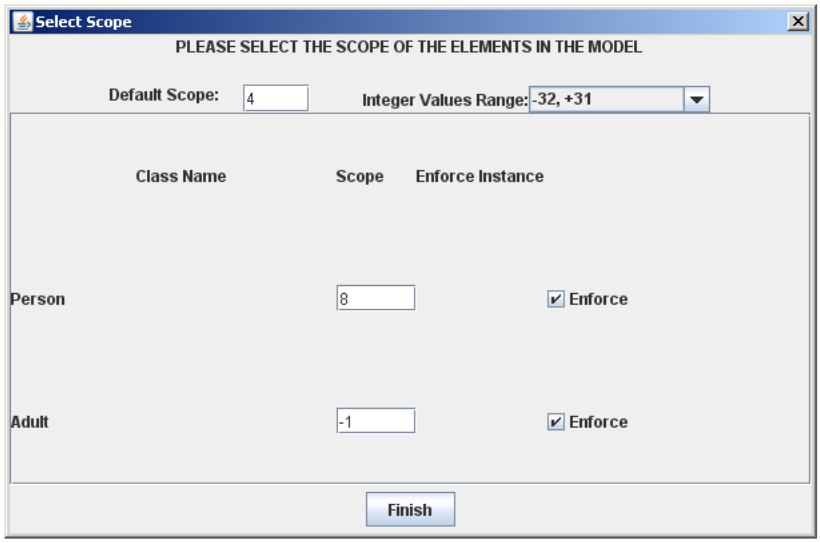

Figure 5.8: Providing the scope

The counterexample shows an Adult, *Adult0* married to an Adult (*Adult1*), whose age is *exactly* 18 years old. This counterexample is produced because according to our OCL invariant an Adult is someone who is 18 years old or older (someone 18 years old is considered to be an Adult), while our assertion checks for people *over* the age of 18.

More specifically the assertion asserts that all Adults (i.e. People 18 years or older) will marry another Adult, whose age is *over* 18 years old. This obviously produces a counterexample.

This small UML example was used as a simple example to show how integer expressions in OCL can be transformed in Alloy. The Alloy counterexample also showed that the Adult, *Adult1* married *Adult0* and *Adult0* is married to *Adult1* (Exercise: In this counterexample *Adult0* is married to *Adult1* and *Adult1* is married to *Adult0*. Is this always the case? Can an *Adult1* be married to *Adult2*, while *Adult2* is married to a third person, *Adult3*?).

```
module untitledModel

some sig Person{
age : one Int}

some sig Adult extends Person{
marry : lone Adult}

fact {My11To01[marry ,Adult ,Adult]}

pred My11To01(r:univ -> univ,
t: set univ, u: set univ){
all x:t|lone y:u|x.r=y
all y:u|one x:t| x.r=y }

fact  { ageInv[  ] }
fact  { adultInv[  ] }

pred adultInv (  ){
all ad :  Adult | int ad . age >= 18
}


pred ageInv (  ){
all p :  Person | int p . age > 0
}

assert assertion {
all ad :  Adult | int ad . marry . age > 18
}

check assertion for 4 but 8 Person, 6 int
```

Figure 5.9: Alloy model of example with integer expressions

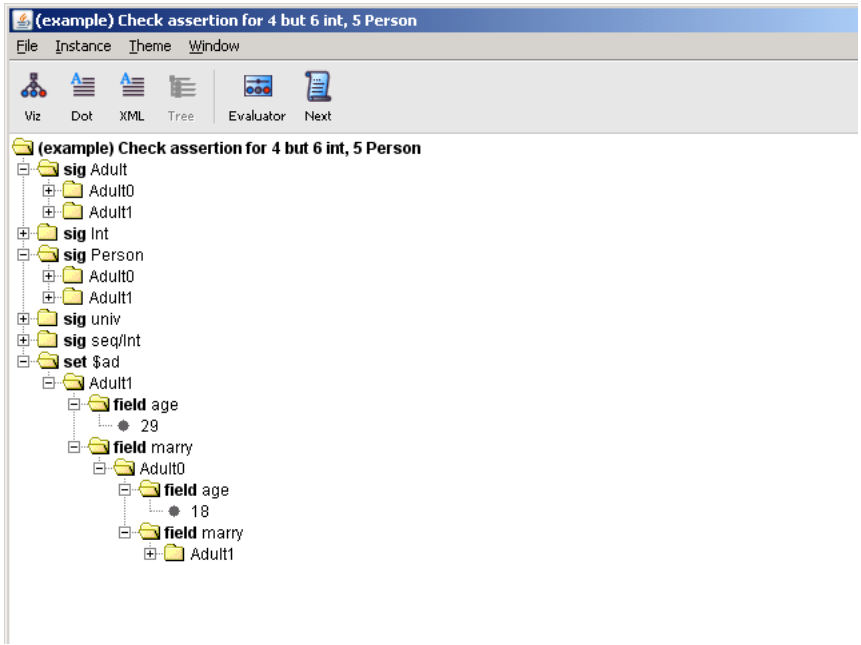

Figure 5.10: A counterexample to the assertion

Appendices

Appendix A XMI Compatibility

We use the KMF XMI parser, so our parser is compatible with the XMI files the KMF parser can read. We have tested UML2Alloy with a number of UML design CASE tools. Table A.1 shows the results of our tests.

CASE TOOL	VERSION	SUCCESS
ArgoUML	0.19	YES
ArgoUML	0.22	YES
Poseidon For UML	4.1 Community Edition	YES
MagicDraw	9.5 Community Edition	YES
MagicDraw	Any Version >= 10	NO

Table A.1: CASE Tools Support

Appendix B UML2Alloy Manual Document Changelog

This page provides an overview of the changes of *this manual document* (not UML2Alloy).

- Sections 3.9 and 5.2, which provide information on the support for Integer Expressions.
- Appendix C on the changelog of the UML2Alloy application.

Appendix C UML2Alloy Application Changelog

- ** RELEASE OF VERSION 0.26 **
- Beta support for XMI 2.1 (XMI produced by the UML plugin of eclipse). It does not support packages. Only supports classes defined in the namespace of the Model. Also only supports OCL expressions defined as OpaqueExpressions.
- 09-Aug-2007: Beta support for integer types and expressions.
- 08-Aug-2007: If an attribute whose type is not supported is found, we raise a fatal exception.
- 08-Aug-2007: If a fatal error occurs the transformation procedure will stop.
- 31-Jul-2007: Compatibility with Alloy v4.
- 25-Jul-2007: If an Association End name has not been specified, by default it will be given the name of the Class (converted to lowercase), which the Association End connects.

Bibliography

[1] Alloy Analyzer website. http://alloy.mit.edu/beta/faq.php [cited April 2005].

[2] UML2Alloy website. http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php.

[3] Martin Gogolla and Mark Richters. Transformation rules for UML class diagrams. In *"UML": '98: Selected papers from the First International Workshop on The Unified Modeling Language "UML": '98*, pages 92–106, London, UK, 1999. Springer-Verlag. ISBN 3-540-66252-9.

[4] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, London, England, 2006.

[5] OMG. OCL Version 2.0, . Document id: formal/06-05-01. http://www.omg.org.

[6] OMG. UML Infrastructure, . Document: formal/05-07-05. http://www.omg.org.

[7] OMG. UML: Superstructure. Version 2.0, . URL http://www.omg.org/cgi-bin/doc?formal/05-07-04. Document id: formal/05-07-04. http://www.omg.org.